numpy_nn

August 22, 2022

[1]:
```python
import os
import pandas as pd
import numpy as np
```

[ ]:
```python
# data = pd.read_csv('')
```

[ ]:
```python
data
```

[ ]:
```python
# split columns into number columns and columns for classification
data_numbers = data.select_dtypes(include=np.number).columns.to_list()
data_names = data.select_dtypes(exclude=np.number).columns.to_list()
```

[ ]:
```python
class Encoders:
    def __init__(self, data, data_names):
        self.data = data
        self.data_names = data_names

    def ordinal_encoder(self):
        for i in self.data_names:
            all_values_list = self.data[f'{i}'].values.tolist()
            values_list = list(dict.fromkeys(all_values_list))
        for k in range(0, len(values_list)):
            self.data.loc[data[f'{i}'] == values_list[k], f'{i}'] = k
        return self.data.astype('int')

encoder = Encoders(data, data_names)
```

[ ]:
```python
encoder.ordinal_encoder()
```

[ ]:
```python
class Activators:
    def __init__(self):
        pass

    def relu(self, z):
        return np.maximum(0, z)

    def drelu(self, z):
        select = (z >= 1)
```

```
        z[select] = 1
        sel = (z < 0)
        z[sel] = 0
        return z

    def sigmoid(self, z):
        return 1 / (1 + np.exp(-z))

    def dsigmoid(self, z):
        return ((2 * np.exp(-z) + 1) / (1 + np.exp(-z))**2)

activator = Activators()
```

[ ]:
```
def normalized_data(data, data_numbers, data_names): # sunormuoja skaitines␣
↪vertes
    normalized_all_lists = []
    for i in data_numbers:
        numbers_list = list(data[f'{i}'])
        normalized_list = [(k / (max(data[f'{i}']) - min(data[f'{i}']))) -␣
↪(min(data[f'{i}']) / (max(data[f'{i}']) - min(data[f'{i}']))) for k in␣
↪numbers_list]
        normalized_all_lists.append(normalized_list)
    for j in data_names:
        names_list = list(data[f'{j}'])
        normalized_all_lists.append(names_list)
    transposed_array = np.array(normalized_all_lists).T # grazina transposed␣
↪array kad stulepliai su eilutem susikeistu vietom
    return pd.DataFrame(transposed_array, columns=data.columns)
```

[ ]:
```
normalized_data = normalized_data(data, data_numbers, data_names)
```

[ ]:
```
normalized_data
```

[ ]:
```
def layers_units():
    layers_number = input('Neural Network layers number (including input and␣
↪output): ')
    layer_units_dic = {'l1': normalized_data_without_output.shape[1]}
    for i in range(2, int(layers_number)):
        l = input(f'{i} layer units: ')
        layer_units_dic.update({f'l{i}': int(l)})
    layer_units_dic.update({f'l{int(layers_number)}': 1})
    batch_size = input('Batch size: ')
    learning_rate = input('Learning rate: ')
    out_col = input('Output column name: ')
    activator_hidden = input('Choose hidden layers activator, type relu or␣
↪softmax: ')
```

```
    activator_output = input('Choose output layer activator, type relu or␣
↪softmax: ')
    return layers_number, layer_units_dic, batch_size, learning_rate, out_col,␣
↪activator_hidden, activator_output
```

[ ]:
```
layers_number, layer_units_dic, batch_size, learning_rate, out_col,␣
↪activator_hidden, activator_output = layers_units()
```

[ ]:
```
normalized_data_without_output = normalized_data.drop(columns=[out_col])
output_data = normalized_data[out_col]
```

[ ]:
```
class NeuralNetwork:
    def __init__(self, input_data, output_data, layers_number, layer_units_dic,␣
↪batch_size, activator_hidden, activator_output):
        self.data = np.array(normalized_data_without_output)
        self.output_data = np.array(output_data).reshape(1, output_data.
↪shape[0])
        self.layers_number = layers_number
        self.layer_units_dic = layer_units_dic
        self.batch_size = int(batch_size)
        self.activator_hidden = activator_hidden
        self.activator_output = activator_output

    def layer_units(self):
        dic_values_list = list(self.layer_units_dic.values())
        dic_values_list.pop(0)
        dic_values_list.pop(-1)
        return dic_values_list

    def weights(self):
        layer_units = self.layer_units()
        dic = {'W1': np.random.randn(self.data.shape[1], layer_units[0])}
        for i in range(2, self.layers_number-1):
            W_arrays_list = list(dic.values())
            dic.update({f'W{i}': np.random.randn(W_arrays_list[i-2].shape[1],␣
↪layer_units[i-1])})
        W_arrays_list = list(dic.values())
        dic.update({f'W{self.layers_number-1}': np.random.
↪randn(W_arrays_list[-1].shape[1], 1)})
        weights_list = [i for i in dic.values()]
        return weights_list

    def bias(self):
        weights_list = self.weights()
        bias_list = []
        for i in range(0, self.layers_number-1):
            bias_list.append(np.random.randn(1, weights_list[i].shape[1]))
```

```
        assert bias_list[-1].shape[1] == weights_list[-1].shape[1]
        return bias_list

    def forward_propagation(self, weights_list, bias_list):
        assert weights_list[0].shape[1] == bias_list[0].shape[1]
        z_list = [self.data.dot(weights_list[0]) + bias_list[0]]
        f_list = [activator.relu(z_list[0])]
        for i in range(1, self.layers_number-2):
            if self.activator_hidden == 'relu':
                z = f_list[i-1].dot(weights_list[i]) + bias_list[i]
                z_list.append(z)
                f = activator.relu(z)
                f_list.append(f)
            else:
                z = f_list[i-1].dot(weights_list[i]) + bias_list[i]
                z_list.append(z)
                f = activator.sigmoid(z)
                f_list.append(f)
        if self.activator_output == 'relu':
            z_list.append(f_list[-1].dot(weights_list[-1]) + bias_list[-1])
            f_list.append(activator.relu(z_list[-1]))
        else:
            z_list.append(f_list[-1].dot(weights_list[-1]) + bias_list[-1])
            f_list.append(activator.sigmoid(z_list[-1]))
        z_list = [i.T for i in z_list]
        f_list = [i.T for i in f_list]
        return z_list, f_list

    def backward_propagation(self, z_list, f_list, weights_list, bias_list,␣
↪batch_size):
        assert len(z_list) == len(f_list) == len(weights_list) == len(bias_list)
        df_list = [f_list[-1] - self.output_data]
        dz_list = []
        dW_list = []
        db_list = []
        for i in range(0, self.layers_number-2):
            if self.activator_hidden == 'relu':
                dz = df_list[i] * activator.drelu(z_list[-(i+1)])
                dz_list.append(dz)
                dW = 1 / self.batch_size * dz_list[i].dot(f_list[-(i+2)].T)
                dW_list.append(dW)
                db = 1 / self.batch_size * np.sum(dz_list[i])
                db_list.append(db)
                df = np.dot(weights_list[-(i+1)], dz_list[i])
                df_list.append(df)
            else:
                dz = df_list[i] * activator.dsigmoid(z_list[-(i+1)])
```

```
                dz_list.append(dz)
                dW = 1 / self.batch_size * dz_list[i].dot(f_list[-(i+2)].T)
                dW_list.append(dW)
                db = 1 / self.batch_size * np.sum(dz_list[i])
                db_list.append(db)
                df = np.dot(weights_list[-(i+1)], dz_list[i])
                df_list.append(df)
        if self.activator_output == 'relu':
            dz_list.append(df_list[-1] * activator.drelu(z_list[0]))
            dW_list.append(1/self.batch_size * dz_list[-1].dot(self.data))
            db_list.append(1/self.batch_size * np.sum(dz_list[-1]))
        else:
            dz_list.append(df_list[-1] * activator.dsigmoid(z_list[0]))
            dW_list.append(1/self.batch_size * dz_list[-1].dot(self.data))
            db_list.append(1/self.batch_size * np.sum(dz_list[-1]))
        assert len(dW_list) == len(db_list)
        new_dW_list = []
        new_db_list = []
        for i in range(1, len(dW_list)+1):
            new_dW_list.append(dW_list[-i])
            new_db_list.append(db_list[-i])
        dW_list = [i.T for i in new_dW_list]
        db_list = [i.T for i in new_db_list]
        return dW_list, db_list

    def update_parameters(self, weights_list, bias_list, dW_list, db_list, lr):
        assert len(dW_list) == len(db_list) == len(weights_list) == 
 ↪len(bias_list)
        assert dW_list[1].shape == weights_list[1].shape
        for i in range(0, len(dW_list)):
            weights_list[i] -= lr * dW_list[i]
            bias_list[i] -= lr * db_list[i]
        return weights_list, bias_list

neural_network = NeuralNetwork(input_data=normalized_data_without_output,
                               output_data=output_data,
                               layers_number=int(layers_number),
                               layer_units_dic=layer_units_dic,
                               batch_size=batch_size,
                               activator_hidden=activator_hidden,
                               activator_output=activator_output)
```

[ ]:
```
class TrainNeuralNetwork:
    def __init__(self, learning_rate, batch_size, output_data):
        self.lr  = learning_rate
        self.batch_size = batch_size
```

```
        self.output_data = np.array(output_data).reshape(1, output_data.
 ↪shape[0])

    def model(self):
        weights_list = neural_network.weights()
        bias_list = neural_network.bias()
        acc_list = []
        loss_list = []
        for i in range(0, 10):
            z_list, f_list = neural_network.forward_propagation(weights_list,␣
 ↪bias_list)
            dW_list, db_list = neural_network.backward_propagation(z_list,␣
 ↪f_list, weights_list, bias_list, self.batch_size)
            weights_list, bias_list = neural_network.
 ↪update_parameters(weights_list, bias_list, dW_list, db_list, self.lr)
            accuracy = (f_list[-1] == self.output_data).all(axis=0).mean()
            acc_list.append(accuracy)
            loss = np.mean((f_list[-1] - self.output_data)**2)
            loss_list.append(loss)
        return acc_list, loss_list

train = TrainNeuralNetwork(learning_rate=float(learning_rate),
                           batch_size=int(batch_size),
                           output_data=output_data)
```

[ ]:
```
train.model()
```